School of Computing and Information Systems
The University of Melbourne
COMP90049
Knowledge Technologies (Semester 1, 2019)
Workshop sample solutions: Week 3

1. Following on from last week, write a **regular expression** which will:

   - Of course, there are numerous ways of writing these. They can also be made far more complicated by dealing with stranger and stranger edge-cases:

   (a) Match a string according to whether it contains a price (like $20 or $0.99, but not 11.30 or 0$n1a).

   - We are using the zero-width word-boundary character `\b` here to indicate that the price-like substring needs to be surrounded by whitespace, or punctuation, or needs to be at the start or end of the string
   - `/\b\$(0|[1-9][0-9]*)(\.\d{1,2})?\b/`

   (b) Match a number in scientific E notation (e.g. `2.00600e+003`)

   - We could make this more complicated by excluding strings like `0.123e0` or `0.000e12`, but a relatively simple solutions would be:
   - `/^(\+|-)?\d(\.\d+)?[eE](\+|-)?\d+$/`

   (c) Remove all HTML comments from an HTML document (defined as a string)

   - The HTML standard is a bit of a moving target. From `https://blog.ostermiller.org/find-comments-html`:
     `s/\<![ \r\n\t]*(--([^\-]|[\r\n]|-[^\-])*--[ \r\n\t]*)\>//g`

   (d) Validate an email address (i.e. the string will match if it is an email address, and will mismatch otherwise)

   - Note that an email address can be a tricky thing to define. See `http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html` for a (long!) Perl regular expression that validates acccording to the RFC 822 grammar (RFC 5322 is too hard for regexes). See a relevant discussion at `http://stackoverflow.com/questions/201323/using-a-regular-expression-to-validate-an-email-address`. A (flawed) example solution from that thread:
     `/^(\w|-)+(\.(\w|-)+)*@(\w|-)+(\.(\w|-)+)*(\.[a-z]{2,4})$/`

---

2. Consider searching the string `muddle-the-middle-muddled-mud` for the query string `led-`:

   (a) How many comparisons are required for the "naive" approach?

   - 30:
     - 1 for each mis-match `m`, `u`, etc.
     - 3 to find the `-` mismatch in `le-`
     - 4 to confirm the found target `led-`

   (b) Identify and construct the extra data structure required to use the version of the Boyer–Moore algorithm discussed in the lecture. (Note that the formal Boyer–Moore algorithm has a richer data structure.) How much extra space is required? How many comparisons within the search string are required? How many operations on the extra data structure?

   - The extra data structure (to the depth that we go to in this subject)[1] is an array of integers, storing the number of positions to jump based on the character observed in the search string.

[1] The actual data structure is an associative array with some longer keys.

- For an ASCII alphabet, this is 256 entries, where the array index is the (unsigned) integer value of the ASCII character (assuming `ord()` can be done in negligible time). Assuming four–byte integers, this is about 256*4B=1KB of memory.
- For this query string, the array stores 1 at position `d`, 2 at `e`, 3 at `l`, there is a match for `-` (a special symbol, which is conventionally 0; at which point we attempt to match the other characters), and 4 for all of the other entries.
- The algorithm works by attempting to match from the end of the query string `led-` (namely, the `-`), which means that the first place we look in the search string is at the fourth position[2], which is a `d`. This isn't a match with `-`, which we determine by looking up the `d` in our data structure, which is associated with a value of 1. Consequently, we move one character to the right in the search string (`l`), and proceed the same way.
- As we can expect random-access lookup into the array, there are 8 array reads: `d` at position 4, `l` at position 5, `t` at 8, `m` at 12, `l` at 16, `m` at 19, `l` at 23, `-` at 26. When we eventually find the `-`, we require 3 more comparisons to confirm the preceding characters are also matches (`led`). This is 11 operations total.

3. Consider compressing the string `muddle-the-middle-muddled-mud`:

   (a) Show the dictionary that would be built using the simple form of LZ coding shown in lectures. Then show the final encoded string, using the lecture notation.

   - We'll build a "dictionary" out of the characters that appear in the original string: `dehilmtu-`. (These are in alpha order (of a sort), but don't need to be. The original dictionary ordering matters less when compressing strings of a non-trivial length.)
   - We build up an LZ message by making reference to the dictionary. We start by pretending that the dictionary is at the start of the string: `dehilmtu-`◇`muddle-the-middle-muddled-mud` (note that the diamond here is not a character, it's just a memory aid for us to know where the dictionary ends and the string to compressed begins. Notably, we **don't count it** below.) Then, for each entry in the target string, we record the number of positions back into the dictionary for the closest match, check the length of the match in the dictionary, and then move the matched substring from the target string to the end of the pretend dictionary.
   - For example, for the first character of the target string `m`, the closest instance in the dictionary is 4 positions back, and the substring match has length 1 (because `mt` in the dictionary is not equal to `mu` in the target). Now the string looks like: `dehilmtu-m`◇`uddle-the-middle-muddled-mud`, and the `m` will be encoded as (4,1).
   - The next character in the target string (using the diamond as our memory aid) is `u`, which is 3 places back in the dictionary. The substring match is again one character, because the string has `ud` but the dictionary has `u-`. `u` will be encoded as (3,1).
   - We continue this same process; each character only matches a single character (based on the closest position in the dictionary heuristic) until we reach the `e` at position 10: `dehilmtu-muddle-th`◇`e-middle-muddled-mud`. Here, the closest `e` in the dictionary is followed by `-`, which is exactly what we need to compress. This is a match of length 2, because `e-` in the dictionary is followed by `t`, but in the string `e-` is followed by `m`. These two characters will be encoded as (4,2), and then we will move our memory-aid forward by two positions, so that the next character to encode is `m`.
   - Eventually, the full encoded string is:
     (4,1)(3,1)(11,1)(1,1)(9,1)(13,1)(7,1)(10,1)(15,1)(4,2)(11,1)(18,1)(10,1)(1,1)(11,3)(7,1)(18,5)(3,1)(8,4)

[2] I am describing these as 1-indexed values, where the first character of the string is at position 1. For most programming languages, the first character in the string is actually at position 0.